C++ Reference Card

C++ Data Types

Data Type	Description
`bool`	boolean (true or false)
`char`	character ('a', 'b', etc.)
`char[]`	character array (C-style string if `null` terminated)
`string`	C++ string (from the STL)
`int`	integer (1, 2, -1, 1000, etc.)
`long int`	long integer
`float`	single precision floating point
`double`	double precision floating point

These are the most commonly used types; this is not a complete list.

Operators

The most commonly used operators in order of precedence:

1	++ (post-increment), -- (post-decrement)
2	! (not), ++ (pre-increment), -- (pre-decrement)
3	*, /, % (modulus)
4	+, -
5	<, <=, >, >=
6	== (equal-to), != (not-equal-to)
7	&& (and)
8	|| (or)
9	= (assignment), *=, /=, %=, +=, -=

Console Input/Output

`cout <<`	console out, printing to screen
`cin >>`	console in, reading from keyboard
`cerr <<`	console error

Example:

```
cout << "Enter an integer: ";
cin >> i;
cout << "Input: " << i << endl;
```

File Input/Output

Example (input):

```
ifstream inputFile;
inputFile.open("data.txt");
inputFile >> inputVariable;
// you can also use get (char) or
// getline (entire line) in addition to >>
  ...
inputFile.close();
```

Example (output):

```
ofstream outFile;
outfile.open("output.txt");
outFile << outputVariable;
  ...
outFile.close();
```

Decision Statements

if

```
if (expression)
    statement;
```

Example

```
if (x < y)
    cout << x;
```

if / else

```
if (expression)
    statement;
else
    statement;
```

Example

```
if (x < y)
    cout << x;
else
    cout << y;
```

switch / case

```
switch(int expression)
{
  case int-constant:
    statement(s);
    break;
  case int-constant:
    statement(s);
    break;
  default:
    statement;
}
```

Example

```
switch(choice)
{
  case 0:
    cout << "Zero";
    break;
  case 1:
    cout << "One";
    break;
  default:
    cout << "What?";
}
```

Looping

while Loop

```
while (expression)
    statement;

while (expression)
{
    statement;
    statement;
}
```

Example

```
while (x < 100)
    cout << x++ << endl;

while (x < 100)
{
    cout << x << endl;
    x++;
}
```

do-while Loop

```
do
    statement;
while (expression);

do
{
    statement;
    statement;
}
while (expression);
```

Example

```
do
    cout << x++ << endl;
while (x < 100);

do
{
    cout << x << endl;
    x++;
}
while (x < 100);
```

for Loop

```
for (initialization; test; update)
    statement;

for (initialization; test; update)
{
    statement;
    statement;
}
```

Example

```
for (count = 0; count < 10; count++)
{
    cout << "count equals: ";
    cout << count << endl;
}
```

Functions

Functions return at most one value. A function that does not return a value has a return type of `void`. Values needed by a function are called parameters.

```
return_type function(type p1, type p2, ...)
{
    statement;
    statement;
    ...
}
```

Examples

```
int timesTwo(int v)
{
    int d;
    d = v * 2;
    return d;
}

void printCourseNumber()
{
    cout << "CSE1284" << endl;
    return;
}
```

Passing Parameters by Value

```
return_type function(type p1)
```

Variable is passed into the function but changes to *p1* are not passed back.

Passing Parameters by Reference

```
return_type function(type &p1)
```

Variable is passed into the function and changes to *p1* are passed back.

Default Parameter Values

```
return_type function(type p1=val)
```

val is used as the value of *p1* if the function is called without a parameter.

Pointers

A pointer variable (or just pointer) is a variable that stores a memory address. Pointers allow the indirect manipulation of data stored in memory.

Pointers are declared using *. To set a pointer's value to the address of another variable, use the & operator.

Example

```
char c = 'a';
char* cPtr;
cPtr = &c;
```

Use the indirection operator (*) to access or change the value that the pointer references.

Example

```
// continued from example above
*cPtr = 'b';
cout << *cPtr << endl;  // prints the char b
cout << c << endl;      // prints the char b
```

Array names can be used as constant pointers, and pointers can be used as array names.

Example

```
int numbers[]={10, 20, 30, 40, 50};
int* numPtr = numbers;
cout << numbers[0] << endl;     // prints 10
cout << *numPtr << endl;        // prints 10
cout << numbers[1] << endl;     // prints 20
cout << *(numPtr + 1) << endl;  // prints 20
cout << numPtr[2] << endl;      // prints 30
```

Dynamic Memory

Allocate Memory

```
ptr = new type;

ptr = new type[size];
```

Examples

```
int* iPtr;
iPtr = new int;

int* intArray;
intArray = new int[5];
```

Deallocate Memory

```
delete ptr;
delete [] ptr;
```

Examples

```
delete iPtr;
delete [] intArray;
```

Once a pointer is used to allocate the memory for an array, array notation can be used to access the array locations.

Example

```
int* intArray;
intArray = new int[5];
intArray[0] = 23;
intArray[1] = 32;
```

Structures

Declaration

```
struct name
{
  type1 element1;
  type2 element2;
};
```

Example

```
struct Hamburger
{
  int patties;
  bool cheese;
};
```

Definition

```
name varName;

name* ptrName;
```

Example

```
Hamburger h;

Hamburger* hPtr;
hPtr = &h;
```

Accessing Members

```
varName.element=val;

ptrName->element=val;
```

Example

```
h.patties = 2;
h.cheese = true;

hPtr->patties = 1;
hPtr->cheese = false;
```

Structures can be used just like the built-in data types in arrays.

Classes

Declaration
```
class classname
{
public:
  classname(params);
  ~classname();
  type member1;
  type member2;
protected:
  type member3;
private:
  type member4;
};
```

Example
```
class Square
{
public:
  Square();
  Square(float w);
  void setWidth(float w);
  float getArea();
private:
  float width;
};
```

public members are accessible from anywhere the class is visible.

private members are only accessible from the same class or a friend (function or class).

protected members are accessible from the same class, derived classes, or a friend (function or class).

constructors may be overloaded just like any other function. You can define two or more constructors as long as each constructor has a different parameter list.

Definition of Member Functions
```
return_type classname::functionName(params)
{
    statements;
}
```

Examples
```
Square::Square()
{
    width = 0;
}

void Square::setWidth(float w)
{
    if (w >= 0)
      width = w;
    else
      exit(-1);
}

float Square::getArea()
{
    return width*width;
}
```

Definition of Instances
```
classname varName;

classname* ptrName;
```

Example
```
Square s1();
Square s2(3.5);

Square* sPtr;
sPtr=new Square(1.8);
```

Accessing Members
```
varName.member=val;
varName.member();

ptrName->member=val;
ptrName->member();
```

Example
```
s1.setWidth(1.5);
cout << s.getArea();

cout<<sPtr->getArea();
```

Inheritance

Inheritance allows a new class to be based on an existing class. The new class inherits all the member variables and functions (except the constructors and destructor) of the class it is based on.

Example
```
class Student
{
public:
  Student(string n, string id);
  void print();
protected:
  string name;
  string netID;
};

class GradStudent : public Student
{
public:
  GradStudent(string n, string id,
                string prev);
  void print();
protected:
  string prevDegree;
};
```

Visibility of Members after Inheritance

Inheritance Specification	Access Specifier in Base Class		
	private	protected	public
private	-	*private*	*private*
protected	-	*protected*	*protected*
public	-	*protected*	*public*

Operator Overloading

C++ allows you to define how standard operators (+, -, *, etc.) work with classes that you write. For example, to use the operator + with your class, you would write a function named `operator+` for your class.

Example
Prototype for a function that overloads + for the `Square` class:
```
Square operator+ (const Square &);
```

If the object that receives the function call is not an instance of a class that you wrote, write the function as a friend of your class. This is standard practice for overloading << and >>.

Example
Prototype for a function that overloads << for the `Square` class:
```
friend ostream & operator<<
          (ostream &, const Square &);
```

Make sure the return type of the overloaded function matches what C++ programmers expect. The return type of relational operators (<, >, ==, etc.) should be `bool`, the return type of << should be `ostream &`, etc.

Exceptions

Example
```
try
{
  // code here calls functions that might
  // throw exceptions
  quotient = divide(num1, num2);

  // or this code might test and throw
  // exceptions directly
  if (num3 < 0)
    throw -1;  // exception to be thrown can
               // be a value or an object
}
catch (int)
{
  cout << "num3 can not be negative!";
  exit(-1);
}
catch (char* exceptionString)
{
  cout << exceptionString;
  exit(-2);
}
//  add more catch blocks as needed
```

Function Templates

Example
```
template <class T>
T getMax(T a, T b)
{
  if (a>b)
    return a;
  else
    return b;
}


// example calls to the function template
int a=9, b=2, c;
c = getMax(a, b);

float f=5.3, g=9.7, h;
h = getMax(f, g);
```

Class Templates

Example
```
template <class T>
class Point
{
public:
  Point(T x, T y);
  void print();
  double distance(Point<T> p);
private:
  T x;
  T y;
};


// examples using the class template
Point<int> p1(3, 2);
Point<float> p2(3.5, 2.5);
p1.print();
p2.print();
```

Suggested Websites

C++ Reference: http://www.cppreference.com/ http://www.informit.com/guides/guide.aspx?g=cplusplus

C++ Tutorial: http://www.cplusplus.com/doc/tutorial/ http://www.sparknotes.com/cs/

C++ Examples: http://www.fredosaurus.com/notes-cpp/

Gaddis Textbook:
- Video Notes http://media.pearsoncmg.com/aw/aw_gaddis_sowcso_6/videos
- Source Code ftp://ftp.aw.com/cseng/authors/gaddis/CCSO5 (5th edition)

C++ QUICK REFERENCE

PREPROCESSOR

```
                        // Comment to end of line
                        /* Multi-line comment */
#include <stdio.h>      // Insert standard header file
#include "myfile.h"     // Insert file in current directory
#define X some text     // Replace X with some text
#define F(a,b) a+b      // Replace F(1,2) with 1+2
#define X \
  some text             // Line continuation
#undef X                // Remove definition
#if defined(X)          // Condional compilation (#ifdef X)
#else                   // Optional (#ifndef X or #if !defined(X))
#endif                  // Required after #if, #ifdef
```

LITERALS

```
255, 0377, 0xff          // Integers (decimal, octal, hex)
2147483647L, 0x7fffffffl // Long (32-bit) integers
123.0, 1.23e2            // double (real) numbers
'a', '\141', '\x61'      // Character (literal, octal, hex)
'\n', '\\', '\'', '\"'   // Newline, backslash, single quote, double
quote
"string\n"               // Array of characters ending with newline and
\0
"hello" "world"          // Concatenated strings
true, false              // bool constants 1 and 0
```

DECLARATIONS

```
int x;                   // Declare x to be an integer (value undefined)
int x=255;               // Declare and initialize x to 255
short s; long l;         // Usually 16 or 32 bit integer (int may be
either)
char c='a';              // Usually 8 bit character
unsigned char u=255; signed char s=-1;  // char might be either
unsigned long x=0xffffffffL;            // short, int, long are signed
float f; double d;       // Single or double precision real (never
unsigned)
bool b=true;             // true or false, may also use int (1 or 0)
int a, b, c;             // Multiple declarations
int a[10];               // Array of 10 ints (a[0] through a[9])
int a[]={0,1,2};         // Initialized array (or a[3]={0,1,2}; )
int a[2][3]={{1,2,3},{4,5,6}};  // Array of array of ints
char s[]="hello";        // String (6 elements including '\0')
int* p;                  // p is a pointer to (address of) int
char* s="hello";         // s points to unnamed array containing "hello"
void* p=NULL;            // Address of untyped memory (NULL is 0)
int& r=x;                // r is a reference to (alias of) int x
enum weekend {SAT,SUN};  // weekend is a type with values SAT and SUN
enum weekend day;        // day is a variable of type weekend
enum weekend {SAT=0,SUN=1};  // Explicit representation as int
enum {SAT,SUN} day;      // Anonymous enum
typedef String char*;    // String s; means char* s;
```

```
const int c=3;           // Constants must be initialized, cannot assign
to
const int* p=a;          // Contents of p (elements of a) are constant
int* const p=a;          // p (but not contents) are constant
const int* const p=a;    // Both p and its contents are constant
const int& cr=x;         // cr cannot be assigned to change x
```

STORAGE CLASSES

```
int x;                   // Auto (memory exists only while in scope)
static int x;            // Global lifetime even if local scope
extern int x;            // Information only, declared elsewhere
```

STATEMENTS

```
x=y;                     // Every expression is a statement
int x;                   // Declarations are statements
;                        // Empty statement

{                        // A block is a single statement
  int x;                 // Scope of x is from declaration to end of
block
  a;                     // In C, declarations must precede statements
}
if (x) a;                // If x is true (not 0), evaluate a
else if (y) b;           // If not x and y (optional, may be repeated)
else c;                  // If not x and not y (optional)

while (x) a;             // Repeat 0 or more times while x is true

for (x; y; z) a;         // Equivalent to: x; while(y) {a; z;}

do a; while (x);         // Equivalent to: a; while(x) a;

switch (x) {             // x must be int
  case X1: a;            // If x == X1 (must be a const), jump here
  case X2: b;            // Else if x == X2, jump here
  default: c;            // Else jump here (optional)
}
break;                   // Jump out of while, do, or for loop, or switch
continue;                // Jump to bottom of while, do, or for loop
return x;                // Return x from function to caller
try { a; }
catch (T t) { b; }       // If a throws a T, then jump here
catch (...) { c; }       // If a throws something else, jump here
```

FUNCTIONS

```
int f(int x, int);       // f is a function taking 2 ints and returning
int
void f();                // f is a procedure taking no arguments
void f(int a=0);         // f() is equivalent to f(0)
f();                     // Default return type is int
inline f();              // Optimize for speed
f() { statements; }      // Function definition (must be global)
T operator+(T x, T y);   // a+b (if type T) calls operator+(a, b)
T operator-(T x);        // -a calls function operator-(a)
T operator++(int);       // postfix ++ or -- (parameter ignored)
extern "C" {void f();}   // f() was compiled in C
```

Function parameters and return values may be of any type. A function must either be declared or defined before it is used. It may be declared first and defined later. Every program consists of a set of a set of global variable declarations and a set of function definitions (possibly in separate files), one of which must be:

```
int main()  { statements... }     or
int main(int argc, char* argv[]) { statements... }
```

argv is an array of argc strings from the command line. By convention, main returns status 0 if successful, 1 or higher for errors.

Functions with different parameters may have the same name (overloading). Operators except :: . .* ?: may be overloaded. Precedence order is not affected. New operators may not be created.

EXPRESSIONS

Operators are grouped by precedence, highest first. Unary operators and assignment evaluate right to left. All others are left to right. Precedence does not affect order of evaluation, which is undefined. There are no run time checks for arrays out of bounds, invalid pointers, etc.

```
T::X                    // Name X defined in class T
N::X                    // Name X defined in namespace N
::X                     // Global name X

t.x                     // Member x of struct or class t
p->x                    // Member x of struct or class pointed to by p
a[i]                    // i'th element of array a
f(x,y)                  // Call to function f with arguments x and y
T(x,y)                  // Object of class T initialized with x and y
x++                     // Add 1 to x, evaluates to original x (postfix)
x--                     // Subtract 1 from x, evaluates to original x
typeid(x)               // Type of x
typeid(T)               // Equals typeid(x) if x is a T
dynamic_cast<T>(x)      // Converts x to a T, checked at run time
static_cast<T>(x)       // Converts x to a T, not checked
reinterpret_cast<T>(x)  // Interpret bits of x as a T
const_cast<T>(x)        // Converts x to same type T but not const

sizeof x                // Number of bytes used to represent object x
sizeof(T)               // Number of bytes to represent type T
++x                     // Add 1 to x, evaluates to new value (prefix)
--x                     // Subtract 1 from x, evaluates to new value
~x                      // Bitwise complement of x
!x                      // true if x is 0, else false (1 or 0 in C)
-x                      // Unary minus
+x                      // Unary plus (default)
&x                      // Address of x
*p                      // Contents of address p (*&x equals x)
new T                   // Address of newly allocated T object
new T(x, y)             // Address of a T initialized with x, y
new T[x]                // Address of allocated n-element array of T
delete p                // Destroy and free object at address p
delete[] p              // Destroy and free array of objects at p
(T) x                   // Convert x to T (obsolete, use .._cast<T>(x))

x * y                   // Multiply
x / y                   // Divide (integers round toward 0)
x % y                   // Modulo (result has sign of x)

x + y                   // Add, or &x[y]
x - y                   // Subtract, or number of elements from *x to *y

x << y                  // x shifted y bits to left (x * pow(2, y))
x >> y                  // x shifted y bits to right (x / pow(2, y))

x < y                   // Less than
x <= y                  // Less than or equal to
x > y                   // Greater than
x >= y                  // Greater than or equal to

x == y                  // Equals
x != y                  // Not equals

x & y                   // Bitwise and (3 & 6 is 2)

x ^ y                   // Bitwise exclusive or (3 ^ 6 is 5)

x | y                   // Bitwise or (3 | 6 is 7)

x && y                  // x and then y (evaluates y only if x (not 0))

x || y                  // x or else y (evaluates y only if x is false
(0))

x = y                   // Assign y to x, returns new value of x
x += y                  // x = x + y, also -= *= /= <<= >>= &= |= ^=

x ? y : z               // y if x is true (nonzero), else z

throw x                 // Throw exception, aborts if not caught

x , y                   // evaluates x and y, returns y (seldom used)
```

CLASSES

```
class T {               // A new type
private:                // Section accessible only to T's member
functions
protected:              // Also accessable to classes derived from T
public:                 // Accessable to all
  int x;                // Member data
  void f();             // Member function
  void g() {return;}    // Inline member function
  void h() const;       // Does not modify any data members
  int operator+(int y); // t+y means t.operator+(y)
  int operator-();      // -t means t.operator-()
  T(): x(1) {}          // Constructor with initialization list
  T(const T& t): x(t.x) {}  // Copy constructor
  T& operator=(const T& t) {x=t.x; return *this; }  // Assignment operator
  ~T();                 // Destructor (automatic cleanup routine)
  explicit T(int a);    // Allow t=T(3) but not t=3
  operator int() const {return x;}  // Allows int(t)
  friend void i();      // Global function i() has private access
  friend class U;       // Members of class U have private access
  static int y;         // Data shared by all T objects
  static void l();      // Shared code.  May access y but not x
  class Z {};           // Nested class T::Z
  typedef int V;        // T::V means int
};
void T::f() {           // Code for member function f of class T
  this->x = x;}         // this is address of self (means x=x;)
int T::y = 2;           // Initialization of static member (required)
T::l();                 // Call to static member
```

```
struct T {               // Equivalent to: class T { public:
  virtual void f();      // May be overridden at run time by derived
class
  virtual void g()=0; };  // Must be overridden (pure virtual)
class U: public T {};     // Derived class U inherits all members of base
T
class V: private T {};    // Inherited members of T become private
class W: public T, public U {};  // Multiple inheritance
class X: public virtual T {}; // Classes derived from X have base T
directly
```

All classes have a default copy constructor, assignment operator, and destructor, which perform the corresponding operations on each data member and each base class as shown above. There is also a default no-argument constructor (required to create arrays) if the class has no constructors. Constructors, assignment, and destructors do not inherit.

TEMPLATES

```
template <class T> T f(T t);        // Overload f for all types
template <class T> class X {        // Class with type parameter T
  X(T t); };                        // A constructor
template <class T> X<T>::X(T t) {}  // Definition of constructor
X<int> x(3);                        // An object of type "X of int"
template <class T, class U=T, int n=0>  // Template with default
parameters
```

NAMESPACES

```
namespace N {class T {};} // Hide name T
N::T t;                   // Use name T in namespace N
using namespace N;        // Make T visible without N::
```

C/C++ STANDARD LIBRARY

Only the most commonly used functions are listed. Header files without .h are in namespace std. File names are actually lower case.

STDIO.H, CSTDIO (Input/output)

```
FILE* f=fopen("filename", "r");  // Open for reading, NULL (0) if error
  // Mode may also be "w" (write) "a" append, "a+" update, "rb" binary
fclose(f);                // Close file f
fprintf(f, "x=%d", 3);    // Print "x=3"  Other conversions:
  "%5d %u %-8ld"            // int width 5, unsigned int, long left just.
  "%o %x %X %lx"            // octal, hex, HEX, long hex
  "%f %5.1f"                // float or double: 123.000000, 123.0
  "%e %g"                   // 1.23e2, use either f or g
  "%c %s"                   // char, char*
  "%%"                      // %
sprintf(s, "x=%d", 3);    // Print to array of char s
printf("x=%d", 3);        // Print to stdout (screen unless redirected)
fprintf(stderr, ...       // Print to standard error (not redirected)
getc(f);                  // Read one char (as an int) or EOF from f
ungetc(c, f);             // Put back one c to f
getchar();                // getc(stdin);
putc(c, f)                // fprintf(f, "%c", c);
putchar(c);               // putc(c, stdout);
fgets(s, n, f);           // Read line into char s[n] from f.  NULL if EOF
gets(s)                   // fgets(s, INT_MAX, f); no bounds check
fread(s, n, 1, f);        // Read n bytes from f to s, return number read
fwrite(s, n, 1, f);       // Write n bytes of s to f, return number
written
fflush(f);                // Force buffered writes to f
fseek(f, n, SEEK_SET);    // Position binary file f at n
ftell(f);                 // Position in f, -1L if error
rewind(f);                // fseek(f, 0L, SEEK_SET); clearerr(f);
feof(f);                  // Is f at end of file?
ferror(f);                // Error in f?
perror(s);                // Print char* s and error message
clearerr(f);              // Clear error code for f
remove("filename");       // Delete file, return 0 if OK
rename("old", "new");     // Rename file, return 0 if OK
f = tmpfile();            // Create temporary file in mode "wb+"
tmpnam(s);                // Put a unique file name in char s[L_tmpnam]
```

STDLIB.H, CSTDLIB (Misc. functions)

```
atof(s); atol(s); atoi(s);// Convert char* s to float, long, int
rand(), srand(seed);      // Random int 0 to RAND_MAX, reset rand()
void* p = malloc(n);      // Allocate n bytes.  Obsolete: use new
free(p);                  // Free memory.  Obsolete: use delete
exit(n);                  // Kill program, return status n
system(s);                // Execute OS command s (system dependent)
getenv("PATH");           // Environment variable or 0 (system dependent)
abs(n); labs(ln);         // Absolute value as int, long
```

STRING.H, CSTRING (Character array handling functions)

Strings are type char[] with a '\0' in the last element used.

```
strcpy(dst, src);         // Copy string. Not bounds checked
strcat(dst, src);         // Concatenate to dst. Not bounds checked
strcmp(s1, s2);           // Compare, <0 if s1<s2, 0 if s1==s2, >0 if
s1>s2
strncpy(dst, src, n);     // Copy up to n chars, also strncat(), strncmp()
strlen(s);                // Length of s not counting \0
strchr(s,c); strrchr(s,c);// Address of first/last char c in s or 0
strstr(s, sub);           // Address of first substring in s or 0
  // mem... functions are for any pointer types (void*), length n bytes
memmove(dst, src, n);     // Copy n bytes from src to dst
memcmp(s1, s2, n);        // Compare n bytes as in strcmp
memchr(s, c, n);          // Find first byte c in s, return address or 0
memset(s, c, n);          // Set n bytes of s to c
```

CTYPE.H, CCTYPE (Character types)

```
isalnum(c);               // Is c a letter or digit?
isalpha(c); isdigit(c);   // Is c a letter?  Digit?
islower(c); isupper(c);   // Is c lower case?  Upper case?
tolower(c); toupper(c);   // Convert c to lower/upper case
```

MATH.H, CMATH (Floating point math)

```
sin(x); cos(x); tan(x);   // Trig functions, x (double) is in radians
```

```
asin(x); acos(x); atan(x);// Inverses
atan2(y, x);             // atan(y/x)
sinh(x); cosh(x); tanh(x);// Hyperbolic
exp(x); log(x); log10(x); // e to the x, log base e, log base 10
pow(x, y); sqrt(x);       // x to the y, square root
ceil(x); floor(x);        // Round up or down (as a double)
fabs(x); fmod(x, y);      // Absolute value, x mod y
```

TIME.H, CTIME (Clock)

```
clock()/CLOCKS_PER_SEC;   // Time in seconds since program started
time_t t=time(0);         // Absolute time in seconds or -1 if unknown
tm* p=gmtime(&t);         // 0 if UCT unavailable, else p->tm_X where X
is:
  sec, min, hour, mday, mon (0-11), year (-1900), wday, yday, isdst
asctime(p);               // "Day Mon dd hh:mm:ss yyyy\n"
asctime(localtime(&t));   // Same format, local time
```

ASSERT.H, CASSERT (Debugging aid)

```
assert(e);                // If e is false, print message and abort
#define NDEBUG            // (before #include <assert.h>), turn off assert
```

NEW.H, NEW (Out of memory handler)

```
set_new_handler(handler); // Change behavior when out of memory
void handler(void) {throw bad_alloc();}  // Default
```

IOSTREAM.H, IOSTREAM (Replaces stdio.h)

```
cin >> x >> y;             // Read words x and y (any type) from stdin
cout << "x=" << 3 << endl;  // Write line to stdout
cerr << x << y << flush;    // Write to stderr and flush
c = cin.get();              // c = getchar();
cin.get(c);                 // Read char
cin.getline(s, n, '\n');    // Read line into char s[n] to '\n' (default)
if (cin)                    // Good state (not EOF)?
                            // To read/write any type T:
istream& operator>>(istream& i, T& x) {i >> ...; x=...; return i;}
ostream& operator<<(ostream& o, const T& x) {return o << ...;}
```

FSTREAM.H, FSTREAM (File I/O works like cin, cout as above)

```
ifstream f1("filename");  // Open text file for reading
if (f1)                   // Test if open and input available
  f1 >> x;                // Read object from file
f1.get(s);                // Read char or line
f1.getline(s, n);         // Read line into string s[n]
ofstream f2("filename");  // Open file for writing
if (f2) f2 << x;          // Write to file
```

IOMANIP.H, IOMANIP (Output formatting)

```
cout << setw(6) << setprecision(2) << setfill('0') << 3.1; // print
"003.10"
```

STRING (Variable sized character array)

```
string s1, s2="hello";    // Create strings
s1.size(), s2.size();     // Number of characters: 0, 5
s1 += s2 + ' ' + "world"; // Concatenation
s1 == "hello world"       // Comparison, also <, >, !=, etc.
s1[0];                    // 'h'
s1.substr(m, n);          // Substring of size n starting at s1[m]
s1.c_str();               // Convert to const char*
getline(cin, s);          // Read line ending in '\n'
```

VECTOR (Variable sized array/stack with built in memory allocation)

```
vector<int> a(10);        // a[0]..a[9] are int (default size is 0)
a.size();                 // Number of elements (10)
a.push_back(3);           // Increase size to 11, a[10]=3
a.back()=4;               // a[10]=4;
a.pop_back();             // Decrease size by 1
a.front();                // a[0];
a[20]=1;                  // Crash: not bounds checked
a.at(20)=1;               // Like a[20] but throws out_of_range()
for (vector<int>::iterator p=a.begin(); p!=a.end(); ++p)
  *p=0;                   // Set all elements of a to 0
vector<int> b(a.begin(), a.end());  // b is copy of a
vector<T> c(n, x);        // c[0]..c[n-1] init to x
T d[10]; vector<T> e(d, d+10);      // e is initialized from d
```

DEQUE (array/stack/queue)

deque<T> is like vector<T>, but also supports:

```
a.push_front(x);          // Puts x at a[0], shifts elements toward back
a.pop_front();            // Removes a[0], shifts toward front
```

UTILITY (Pair)

```
pair<string, int> a("hello", 3);  // A 2-element struct
a.first;                  // "hello"
a.second;                 // 3
```

MAP (associative array)

```
map<string, int> a;       // Map from string to int
a["hello"]=3;             // Add or replace element a["hello"]
for (map<string, int>::iterator p=a.begin(); p!=a.end(); ++p)
  cout << (*p).first << (*p).second;  // Prints hello, 3
a.size();                 // 1
```

ALGORITHM (A collection of 60 algorithms on sequences with iterators)

```
min(x, y); max(x, y);     // Smaller/larger of x, y (any type defining <)
swap(x, y);               // Exchange values of variables x and y
sort(a, a+n);             // Sort array a[0]..a[n-1] by <
sort(a.begin(), a.end()); // Sort vector or deque
```

C++ Reference Card

Key

switch – keyword, reserved
"Hello!" – string
// comment – commented code
close() – library function
main – variable, identifier
variable – placeholder in syntax

```
if (expression) - syntax
  statement;
```

C++ Program Structure

```
// my first program in C++
#include <iostream.h>
int main ()
{
  cout << "Hello World!";
  return 0;
}

// single line comment
/* multi-line
   comment */
```

Identifiers

These are ANSI C++ reserved words and cannot be used as variable names.

asm, auto, bool, break, case, catch, char, class, const, const_cast, continue, default, delete, do, double, dynamic_cast, else, enum, explicit, extern, false, float, for, friend, goto, if, inline, int, long, mutable, namespace, new, operator, private, protected, public, register, reinterpret_cast, return, short, signed, sizeof, static, static_cast, struct, switch, template, this, throw, true, try, typedef, typeid, typename, union, unsigned, using, virtual, void, volatile, wchar_t

Data Types

Variable Declaration

```
special class size sign type name;
special: volatile
class: register, static, extern, auto
size: long, short, double
sign: signed, unsigned
type: int, float, char (required)
name: the variable name (required)
// example of variable declaration
extern short unsigned char AFlag;
```

TYPE	SIZE	RANGE
char	1	signed -128 to 127
		unsigned 0 to 255
short	2	signed -32,768 to 32,767
		unsigned 0 to 65,535
long	4	signed -2,147,483,648 to 2,147,483,647
		unsigned 0 - 4,294,967,295
int	varies depending on system	
float	4	3.4E +/- 38 (7 digits)
double	8	1.7E +/- 308 (15 digits)
long double	10	1.2E +/- 4,932 (19 digits)
bool	1	true or false
wchar_t	2	wide characters

Pointers

```
type *variable; // pointer to variable
type *func(); // function returns pointer
void * // generic pointer type
NULL;  // null pointer
*ptr;  // object pointed to by pointer
&obj   // address of object
```

Arrays

```
int arry[n];          // array of size n
int arry2d[n][m];     // 2d n x m array
int arry3d[i][j][k]; // 3d ixjxk array
```

Structures

```
struct name {
  type1 element1;
  type2 element2;
  ...
} object_name; // instance of name
name variable; // variable of type name
variable.element1; // ref. of element
variable->element1; // reference of
pointed to structure
```

Initialization of Variables

```
type id;        // declaration
type id,id,id; // multiple declaration
type *id;       // pointer declaration
type id = value;  // declare with assign
type *id = value; // pointer with assign
id = value; // assignment
```

Examples

```
// single character in single quotes
char c='A';
// string in double quotes, ptr to string
char *str = "Hello";
int i = 1022;
float f = 4.0E10; // 4^10
int ary[2] = {1,2} // array of ints
const int a = 45; // constant declaration
struct products { // declaration
  char name [30];
  float price;
} ;
products apple; // create instance
apple.name = "Macintosh"; // assignment
apple.price = 0.45;
products *pApple; // pointer to struct
pApple->name = "Granny Smith";
pApple->price = 0.35; // assignment
```

Exceptions

```
try {
  // code to be tried... if statements
  statements;  // fail, exception is set
  throw exception;
}
catch (type exception) {
  // code in case of exception
  statements;
}
```

Operators

priority/**operator**/**desc**/ASSOCIATIVITY

1 :: **scope** LEFT

2 () **parenthesis** LEFT
[ ] **brackets** LEFT
-> **pointer reference** LEFT
. **structure member access** LEFT
sizeof returns memory size LEFT

3 ++ **increment** RIGHT
-- **decrement** RIGHT
~ **complement to one (bitwise)** RIGHT
! **unary NOT** RIGHT
& **reference (pointers)** RIGHT
* **dereference** RIGHT
(type) **type casting** RIGHT
+ - **unary less sign** RIGHT

4 * **multiply** LEFT
/ **divide** LEFT
% **modulus** LEFT

5 + **addition** LEFT
- **subtraction** LEFT

6 << **bitwise shift left** LEFT
>> **bitwise shift right** LEFT

7 < **less than** LEFT
<= **less than or equal** LEFT
> **greater than** LEFT
>= **greater than or equal** LEFT

8 == **equal** LEFT
!= **not equal** LEFT

9 & **bitwise AND** LEFT
^ **bitwise NOT** LEFT
| **bitwise OR** LEFT

10 && **logical AND** LEFT
|| **logical OR** LEFT

11 ? : **conditional** RIGHT

12 = **assignment**
+= **add/assign**
-= **subtract/assign**
*= **multiply/assign**
/= **divide/assign**
%= **modulus/assign**
>>= **bitwise shift right/assign**
<<= **bitwise shift left/assign**
&= **bitwise AND/assign**
^= **bitwise NOT/assign**
|= **bitwise OR/assign**

13 , **comma**

User Defined DataTypes

```
typedef existingtype newtypename;
typedef unsigned int WORD;
enum name{val1, val2, ...} obj_name;
enum days_t {MON,WED,FRI} days;
union model_name {
  type1 element1;
  type2 element2; ...
} object_name;
union mytypes_t {
  char c;
  int i;
} mytypes;
struct packed {  // bit fields
  unsigned int flagA:1; // flagA is 1 bit
  unsigned int flagB:3; // flagB is 3 bit
}
```

Preprocessor Directives

```
#define ID value // replaces ID with
//value for each occurrence in the code
#undef ID  // reverse of #define
#ifdef ID  //executes code if ID defined
#ifndef ID // opposite of #ifdef
#if expr   // executes if expr is true
#else      // else
#elif      // else if
#endif     // ends if block
#line number "filename"
// #line controls what line number and
// filename appear when a compiler error
// occurs
#error msg //reports msg on cmpl. error
#include "file" // inserts file into code
                 // during compilation
 #pragma //passes parameters to compiler
```

Control Structures

Decision (if-else)

```
if (condition) {
  statements;
}
else if (condition) {
  statements;
}
else {
  statements;
}
if (x == 3)  // curly braces not needed
  flag = 1;  // when if statement is
else         // followed by only one
  flag = 0;  // statement
```

Repetition (while)

```
while (expression) { // loop until
  statements;       // expression is false
}
```

Repetition (do-while)

```
do {          // perform the statements
  statements; // as long as condition
} while (condition); // is true
```

Repetition (for)

init - initial value for loop control variable
condition - stay in the loop as long as condition is true
increment - change the loop control variable

```
for(init; condition; increment) {
  statements;
}
```

Bifurcation (break, continue, goto, exit)

```
break; // ends a loop
continue; // stops executing statements
// in current iteration of loop cont-
// inues executing on next iteration
label:
goto label; // execution continues at
// label
exit(retcode); // exits program
```

Selection (switch)

```
switch (variable) {
  case constant1: // chars, ints
    statements;
    break; // needed to end flow
  case constant2:
    statements;
    break;
  default:
    statements; // default statements
}
```

Console Input/Output

[See ***File I/O*** *on reverse for more about streams*]

C Style Console I/O

```
stdin - standard input stream
stdout - standard output stream
stderr - standard error stream
// print to screen with formatting
printf("format", arg1,arg2,...);
printf("nums: %d, %f, %c", 1,5.6,'C');
// print to string s
sprintf(s,"format", arg1, arg2,...);
sprintf(s,"This is string # %i",2);
// read data from keyboard into
// name1,name2,...
scanf("format",&name1,&name2, ...);
scanf("%d,%f",var1,var2); // read nums
// read from string s
sscanf("format",&name1,&name2, ...);
sscanf(s,"%i,%c",var1,var2);
```

C Style I/O Formatting

```
%d, %i  integer
%c      single character
%f      double (float)
%o      octal
%p      pointer
%u      unsigned
%s      char string
%e, %E  exponential
%x, %X  hexadecimal
%n      number of chars written
%g, %G  same as f for e,E
```

C++ console I/O

```
cout<< console out, printing to screen
cin>> console in, reading from keyboard
cerr<< console error
clog<< console log
cout<<"Please enter an integer: ";
cin>>i;
cout<<"num1: "<<i<<"\n"<<endl;
```

Control Characters

```
\b backspace   \f form feed  \r return
\' apostrophe  \n newline    \t tab
\nnn character #nnn (octal)  \" quote
\NN character #NN (hexadecimal)
```

Character Strings

The string "Hello" is actually composed of 6 characters and is stored in memory as follows:

Char	H	e	l	l	o	\0
Index	0	1	2	3	4	5

\0 (backslash zero) is the null terminator character and determines the end of the string. A string is an array of characters. Arrays in C and C++ start at zero.

```
str = "Hello";
str[2] = 'e';  // string is now 'Heelo'
```

common <string.h> functions:

strcat(*s1*,*s2*) **strchr**(*s1*,*c*) **strcmp**(*s1*,*s2*)
strcpy(*s2*,*s1*) **strlen**(*s1*) **strncpy**(*s2*,*s1*,*n*)
strstr(*s1*,*s2*)

Functions

In C, functions must be prototyped before the main function, and defined after the **main** function. In C++, functions may, but do not need to be, prototyped. C++ functions must be defined before the location where they are called from.

```
// function declaration
type name(arg1, arg2, ...) {
  statement1;
  statement2;
  ...
}
```

type – return type of the function
name – name by which the function is called
arg1, arg2 – parameters to the function
statement – statements inside the function

```
// example function declaration
// return type int
int add(int a, int b) { // parms
  int r;                // declaration
  r = a + b;            // add nums
  return r;             // return value
}
// function call
num = add(1,2);
```

Passing Parameters

Pass by Value

```
function(int var); // passed by value
```

Variable is passed into the function and can be changed, but changes are not passed back.

Pass by Constant Value

```
function(const int var);
```

Variable is passed into the function but cannot be changed.

Pass by Reference

```
function(int &var); // pass by reference
```

Variable is passed into the function and can be changed, changes are passed back.

Pass by Constant Reference

```
function(const int &var);
```

Variable cannot be changed in the function.

Passing an Array by Reference

It's a waste of memory to pass arrays and structures by value, instead pass by reference.

```
int array[1]; // array declaration
ret = aryfunc(&array); // function call
int aryfunc(int *array[1]) {
  array[0] = 2;          // function
  return 2;              // declaration
}
```

Default Parameter Values

```
int add(int a, int b=2) {
  int r;
  r=a+b;     // b is always 2
  return (r);
}
```

Overloading Functions

Functions can have the same name, and same number of parameters as long as the parameters of are different types

```
// takes and returns integers
int divide (int a, int b)
{ return (a/b); }
// takes and returns floats
float divide (float a, float b)
{ return (a/b); }
divide(10,2); // returns 5
divide(10,3); // returns 3.33333333
```

Recursion

Functions can call themselves

```
long factorial (long n) {
  if (n > 1)
    return (n * factorial (n-1));
  else
    return (1);
}
```

Prototyping

Functions can be prototyped so they can be used after being declared in any order

```
// prototyped functions can be used
// anywhere in the program
#include <iostream.h>
void odd (int a);
void even (int a);
int main () { ... }
```

Namespaces

Namespaces allow global identifiers under a name

```
// simple namespace
namespace identifier {
  namespace-body;
}
// example namespace
namespace first {int var = 5;}
namespace second {double var = 3.1416;}
int main () {
  cout << first::var << endl;
  cout << second::var << endl;
  return 0;
}
```

using namespace allows for the current nesting level to use the appropriate namespace

```
using namespace identifier;
// example using namespace
namespace first {int var = 5;}
namespace second {double var = 3.1416;}
int main () {
  using namespace second;
  cout << var << endl;
  cout << (var*2) << endl;
  return 0;
}
```

Class Reference

Class Syntax

```
class classname {
  public:
    classname(parms); // constructor
    ~classname();     // destructor
    member1;
    member2;
  protected:
    member3;
    ...
  private:
    member4;
} objectname;
// constructor (initializes variables)
classname::classname(parms) {
}
// destructor (deletes variables)
classname::~classname() {
}
```

public members are accessible from anywhere where the class is visible
protected members are only accessible from members of the same class or of a friend class
private members are accessible from members of the same class, members of the derived classes and a friend class
constructors may be overloaded just like any other function. define two identical constructors with difference parameter lists

Class Example

```
class CSquare { // class declaration
  public:
    void Init(float h, float w);
    float GetArea(); // functions
  private: // available only to CSquare
    float h,w;
}  // implementations of functions
void CSquare::Init(float hi, float wi){
  h = hi; w = wi;
}
float CSquare::GetArea() {
  return (h*w);
}
// example declaration and usage
CSquare theSquare;
theSquare.Init(8,5);
area = theSquare.GetArea();
// or using a pointer to the class
CSquare *theSquare;
theSquare->Init(8,5);
 area = theSquare->GetArea();
```

Overloading Operators

Like functions, operators can be overloaded. Imagine you have a class that defines a square and you create two instances of the class. You can add the two objects together.

```
class CSquare { // declare a class
  public: // functions
    void Init(float h, float w);
    float GetArea();
    CSquare operator + (CSquare);
  private:   // overload the '+' operator
    float h,w;
} // function implementations
void CSquare::Init(float hi, float wi){
  h = hi; w = wi;
}
float CSquare::GetArea() {
  return (h*w);
}// implementation of overloaded operator
CSquare CSquare::operator+ (CSquare cs) {
  CSquare temp; // create CSquare object
  temp.h = h + cs.h; // add h and w to
  temp.w = w + cs.w; // temp object
  return (temp);
}
// object declaration and usage
CSquare sqr1, sqr2, sqr3;
sqr1.Init(3,4); // initialize objects
sqr2.Init(2,3);
sqr3 = sqr1 + sqr2; // object sqr3 is now
(5,7)
```

Advanced Class Syntax

Static Keyword

static variables are the same throughout all instances of a class.

```
static int n; // declaration
CDummy::n;    // reference
```

Virtual Members

Classes may have virtual members. If the function is redefined in an inherited class, the parent must have the word **virtual** in front of the function definition

This keyword

The **this** keyword refers to the memory location of the current object.

```
int func(this); // passes pointer to
                // current object
```

Class TypeCasting

```
reinterpret_cast <newtype>(expression);
    dynamic_cast <newtype>(expression);
     static_cast <newtype>(expression);
      const_cast <newtype>(expression);
```

Expression Type

The type of an expression can be found using **typeid**. **typeid** returns a type.

```
typeid(expression);
```

Inheritance

Functions from a class can be inherited and reused in other classes. **Multiple inheritance** is possible.

```
class CPoly { //create base polygon class
  protected:
    int width, height;
  public:
    void SetValues(int a, int b)
      { width=a; height=b;}
};
class COutput { // create base output
  public:       // class
    void Output(int i);
};
void COutput::Output (int i) {
  cout << i << endl;
}
// CRect inherits SetValues from Cpoly
// and inherits Output from COutput
class CRect: public CPoly, public COutput
{
  public:
    int area(void)
      { return (width * height); }
};
// CTri inherits SetValues from CPoly
class CTri: public CPoly {
  public:
    int area(void)
      { return (width * height / 2); }
};
void main () {
  CRect rect; // declare objects
  CTri tri;
  rect.SetValues (2,9);
  tri.SetValues (2,9);
  rect.Output(rect.area());
  cout<<tri.area()<<endl;
}
```

Templates

Templates allow functions and classes to be reused without overloading them

```
template <class id> function;
template <typename id> function;
// ---------- function example ---------
template <class T>
T GetMax (T a, T b) {
  return (a>b?a:b); // return the larger
}
void main () {
  int a=9, b=2, c;
  float x=5.3, y=3.2, z;
  c=GetMax(a,b);
  z=GetMax(x,y);
}
// ----------- class example -----------
template <class T>
class CPair {
    T x,y;
  public:
    Pair(T a, T b){
      x=a; y=b; }
    T GetMax();
};
template <class T>
T Pair<T>::GetMax()
{ // implementation of GetMax function
  T ret; // return a template
  ret = x>y?x:y; // return larger
  return ret;
}
int main () {
  Pair <int> theMax (80, 45);
  cout << theMax.GetMax();
  return 0;
}
```

Friend Classes/Functions

Friend Class Example

```
class CSquare;  // define CSquare
class CRectangle {
  int width, height;
  public:
    void convert (CSquare a);
};
class CSquare { // we want to use the
  private:      // convert function in
    int side;   // the CSquare class, so
  public:       // use the friend keyword
    void set_side (int a) { side=a; }
    friend class CRectangle;
};
void CRectangle::convert (CSquare a) {
  width = a.side;
  height = a.side;
}
// declaration and usage
CSquare sqr;
CRectangle rect;   // convert can be
sqr.set_side(4);   // used by the
rect.convert(sqr); // rectangle class
```

Friend Functions

A friend function has the keyword **friend** in front of it. If it is declared inside a class, that function can be called without reference from an object. An object may be passed to it.

```
/* change can be used anywhere and can
have a CRect object passed in */
// this example defined inside a class
friend CRect change(CRect);
CRectangle recta, rectb; // declaration
 rectb = change(recta);  // usage
```

File I/O

```
#include <fstream.h>  // read/write file
#include <ofstream.h> // write file
#include <ifstream.h> // read file
```

File I/O is done from the **fstream**, **ofstream**, and **ifstream** classes.

File Handles

A file must have a file handle (pointer to the file) to access the file.

```
ifstream infile; // create handle called
// infile to read from a file
ofstream outfile; // handle for writing
fstream f; // handle for read/write
```

Opening Files

After declaring a file handle, the following syntax can be used to open the file

```
void open(const char *fname, ios::mode);
```

fname should be a string, specifying an absolute or relative path, including filename. **ios::***mode* can be any number of the following and repeat:

in Open file for reading
out Open file for writing
ate Initial position: end of file
app Every output is appended at the end of file
trunc If the file already existed it is erased
binary Binary mode

```
ifstream f; // open input file example
f.open("input.txt", ios::in);
ofstream f; // open for writing in binary
f.open("out.txt", ios::out | ios::binary
| ios::app);
```

Closing a File

A file can be closed by calling the handle's close function

```
f.close();
```

Writing To a File (Text Mode)

The operator **<<** can be used to write to a file. Like **cout**, a stream can be opened to a device. For file writing, the device is not the console, it is the file. **cout** is replaced with the file handle.

```
ofstream f; // create file handle
f.open("output.txt") // open file
f <<"Hello World\n"<<a<<b<<c<<endl;
```

Reading From a File (Text Mode)

The operator **>>** can be used to read from a file. It works similar to cin. Fields are seperated in the file by spaces.

```
ifstream f; // create file handle
f.open("input.txt"); // open file
while (!f.eof()) // end of file test
  f >>a>>b>>c; // read into a,b,c
```

I/O State Flags

Flags are set if errors or other conditions occur. The following functions are members of the file object

handle.**bad()** returns true if a failure occurs in reading or writing
handle.**fail()** returns true for same cases as bad() plus if formatting errors occur
handle.**eof()** returns true if the end of the file reached when reading
handle.**good()** returns false if any of the above were true

Stream Pointers

handle.**tellg()** returns pointer to current location when reading a file
handle.**tellp()** returns pointer to current location when writing a file

```
// seek a position in reading a file
handle.seekg(position);
handle.seekg(offset, direction);
// seek a position in writing a file
handle.seekp(position);
handle.seekp(offset, direction);
```

direction can be one of the following

ios::beg beginning of the stream
ios::cur current position of the stream pointer
ios::end end of the stream

Binary Files

buffer is a location to store the characters. *numbytes* is the number of bytes to written or read.

```
write(char *buffer, numbytes);
read(char *buffer, numbytes);
```

Output Formatting

```
streamclass f;  // declare file handle
// set output flags
f.flags(ios_base::flag)
```

possible *flags*

dec	**fixed**	**hex**	**oct**
scientific	**internal**	**left**	**right**
uppercase	**boolalpha**	**showbase**	**showpoint**
showpos	**skipws**	**unitbuf**	

adjustfield left | right | internal
basefield dec | oct | hex
floatfield scientific | fixed

```
f.fill() get fill character
f.fill(ch) set fill character ch
f.precision(numdigits) sets the precision for
floating point numbers to numdigits
f.put(c) put a single char into output stream
f.setf(flag) sets a flag
f.setf(flag, mask) sets a flag w/value
f.width() returns the current number of
characters to be written
f.width(num) sets the number of chars to be
written
```

ACSII Chart

Dec	Char	Dec	Char	Dec	Char	Dec	Char
0	NUL	64	@	128	Ç	192	└
1	SOH	65	A	129	ü	193	┴
2	STX	66	B	130	é	194	┬
3	ETX	67	C	131	â	195	├
4	EOT	68	D	132	ä	196	─
5	ENQ	69	E	133	à	197	┼
6	ACK	70	F	134	å	198	╞
7	BEL	71	G	135	ç	199	╟
8	BS	72	H	136	ê	200	╚
9	TAB	73	I	137	ë	201	╔
10	LF	74	J	138	è	202	╩
11	VTB	75	K	139	ï	203	╦
12	FF	76	L	140	î	204	╠
13	CR	77	M	141	ì	205	═
14	SO	78	N	142	Ä	206	╬
15	SI	79	O	143	Å	207	╧
16	DLE	80	P	144	É	208	╨
17	DC1	81	Q	145	æ	209	╤
18	DC2	82	R	146	Æ	210	╥
19	DC3	83	S	147	ô	211	╙
20	DC4	84	T	148	ö	212	╘
21	NAK	85	U	149	ò	213	╒
22	SYN	86	V	150	û	214	╓
23	ETB	87	W	151	ù	215	╫
24	CAN	88	X	152	ÿ	216	╪
25	EM	89	Y	153	Ö	217	┘
26	SUB	90	Z	154	Ü	218	┌
27	ESC	91	[	155	¢	219	■
28	FS	92	\	156	£	220	▄
29	GS	93	]	157	¥	221	▌
30	RS	94	^	158	?	222	?
31	US	95	_	159	ƒ	223	?
32		96	`	160	á	224	α
33	!	97	a	161	í	225	ß
34	"	98	b	162	ó	226	Γ
35	#	99	c	163	ú	227	π
36	$	100	d	164	ñ	228	Σ
37	%	101	e	165	Ñ	229	σ
38	&	102	f	166	ª	230	µ
39	'	103	g	167	º	231	τ
40	(	104	h	168	¿	232	Φ
41	)	105	i	169	?	233	Θ
42	*	106	j	170	¬	234	Ω
43	+	107	k	171	½	235	δ
44	,	108	l	172	¼	236	∞
45	-	109	m	173	¡	237	φ
46	.	110	n	174	«	238	ε
47	/	111	o	175	»	239	∩
48	0	112	p	176	?	240	≡
49	1	113	q	177	▒	241	±
50	2	114	r	178	▓	242	≥
51	3	115	s	179	│	243	≤
52	4	116	t	180	┤	244	?
53	5	117	u	181	┤	245	?
54	6	118	v	182	┤	246	÷
55	7	119	w	183	┐	247	≈
56	8	120	x	184	┐	248	°
57	9	121	y	185	┤	249	?
58	:	122	z	186	┃	250	·
59	;	123	{	187	┐	251	√
60	<	124	|	188	┘	252	ⁿ
61	=	125	}	189	┘	253	²
62	>	126	~	190	┘	254	■
63	?	127	?	191	┐	255	

Dynamic Memory

Memory can be allocated and deallocated

```
// allocate memory (C++ only)
pointer = new type [];
int *ptr;          // declare a pointer
ptr = new int;     // create a new instance
ptr = new int [5]; // new array of ints
// deallocate memory (C++ only)
delete [] pointer;
delete ptr;    // delete a single int
delete [] ptr // delete array
// allocate memory (C or C++)
void * malloc (nbytes); // nbytes=size
char *buffer;  // declare a buffer
// allocate 10 bytes to the buffer
buffer = (char *)malloc(10);
// allocate memory (C or C++)
// nelements = number elements
// size = size of each element
void * malloc (nelements, size);
int *nums;  // declare a buffer
// allocate 5 sets of ints
nums = (char *)calloc(5,sizeof(int));
// reallocate memory (C or C++)
void * realloc (*ptr, size);
// delete memory (C or C++)
 void free (*ptr);
```

ANSI C++ Library Files

The following files are part of the ANSI C++ standard and should work in most compilers.

```
<algorithm.h> <bitset.h> <deque.h>
<exception.h> <fstream.h> <functional.h>
<iomanip.h> <ios.h> <iosfwd.h>
<iostream.h> <istream.h> <iterator.h>
<limits.h> <list.h> <locale.h> <map.h>
<memory.h> <new.h> <numeric.h>
<ostream.h> <queue.h> <set.h> <sstream.h>
<stack.h> <stdexcept.h> <streambuf.h>
<string.h> <typeinfo.h> <utility.h>
<valarray.h> <vector.h>
```


C++ Reference Card
C/C++ Syntax, DataTypes, Functions
Classes, I/O Stream Library Functions

refcard@gbook.org